

MODELING AND REPRESENTATION OF THREE-DIMENSIONAL OBJECTS

Many structures and methods can represent three-dimensional objects in computer graphics. The choice depends on several factors. The *shape and type of the object itself* are important, as is the *purpose* of the representation. If the goal is to produce an attractive image, an approximate representation of the object is usually sufficient, and a polygon mesh is a common choice. When designing a mechanical component or a similar object, an approximate polygon mesh is inadequate. In such cases, exact models such as a CSG representation (for example, in mechanical engineering) or a representation using spline surfaces (for modeling car bodies or aircraft fuselages) are more appropriate. The *form of the input data* also plays an important role. A typical example is the processing and visualization of medical data. When the input is already represented by voxels, considering a different representation is usually unnecessary. The user (a physician) must extract the required information from the raster structure.

An object representation must be chosen both for the user interface, where new objects are created and existing ones are modified, and for storing data inside the graphics system. These two representations may, but need not, be identical. For example, it is unrealistic to expect a user to enter an object represented by a massive polygon mesh directly.

The most widely used object representations, ordered by popularity, are:

- polygon meshes
- splines
- CSG representations
- space-subdivision representations
 - octrees
 - BSP trees
- implicit representations

1. POLYGON MESHES

This is a boundary representation, which means that the boundary of the object is represented. Polygon meshes are clearly the dominant representation in modern computer graphics. They are very simple and do not require a deeper analysis of the objects. The surface is merely approximated by a collection of polygons. On the other hand, this is clearly a very inefficient way to store information about an object. The higher the required accuracy, the larger the data set that must be stored, and all structural information (symmetry, smoothness, etc.) is lost. Paradoxically, algorithms that process objects represented in this way are very fast. The reason is precisely the popularity of meshes, which has encouraged the development of specialized algorithms, their optimization, and even hardware support for mesh-based objects. Consequently, other representations are often converted to polygon meshes, for example for visualization.

Polygon meshes are also frequently used in hybrid representations, for example together with photographic textures. In interactive applications such as games and virtual reality, speed requirements often lead to a very coarse approximation of the object onto which a texture is mapped. The texture then becomes part of the model rather than merely an attribute used during visualization. Shading algorithms also usually assume that an object represented by a polygon mesh, that is, a surface with edges, is actually smooth. Sharp edges are therefore smoothed during shading.

1.1. Basic Concepts.

Definition 1.1. A *polygon mesh* is a collection of *vertices* (i.e., a finite set of points), *edges* (i.e., line segments whose endpoints are vertices), and *faces* (i.e., convex planar polygons whose boundaries consist of edges), subject to the following conditions:

- every vertex is an endpoint of at least one edge,
- every edge is part of the boundary of at least one face,
- if two faces have a nonempty intersection, the intersection is a mesh element (a vertex or an edge).

Triangle meshes, in which every face is a triangle, are used most often. Some graphics systems even support only such meshes. This is not a fundamental limitation because every polygon can be triangulated, even without adding new vertices. Triangle meshes also avoid problems with face planarity. Three vertices always lie in a plane, whereas a face defined by more vertices need not be planar. Deviations from a plane may arise, for example, because of rounding.

Definition 1.2. The *boundary* of a mesh is the union of the edges that belong to only one face. A polygon mesh is *closed* if it has no boundary.

Definition 1.3. A polygon mesh is called *connected* if the graph obtained by replacing mesh faces with graph vertices is connected, with two graph vertices joined by an edge whenever the corresponding faces share a mesh edge.

Definition 1.4. A polygon mesh is called a *manifold* if every interior point of the mesh has a neighborhood topologically isomorphic to a neighborhood of a point in the plane. (The neighborhood of a mesh point means the intersection of its spatial neighborhood with the mesh.)

If a mesh is a manifold, each edge forms the boundary of at most two faces. (A condition on the vertices also follows, but it is more difficult to describe informally.)

Definition 1.5. An *orientation* of a polygon is chosen by listing its vertices along the boundary. Two adjacent polygons, that is, polygons sharing an edge, are oriented *consistently* if the vertices along the common edge are listed in opposite orders in the two polygons. A polygon mesh that is a manifold is called *orientable* if each polygon can be assigned an orientation such that every pair of adjacent polygons is oriented consistently.

1.2. Data Structures. The simplest way to represent a polygon mesh is to list its individual elements. We therefore use a table or list of vertices that stores the coordinates of each vertex, an edge table that stores pointers, indices, or names of the endpoints of each edge, and a face table that stores the boundary edges of each face (as pointers, names, etc.).

This representation is useful for many purposes. It contains all the information needed to describe the mesh and imposes no restrictions on its type or properties, such as orientability or the manifold property. However, modifying a mesh becomes cumbersome when the changes are more complex than updating the coordinates of individual vertices. The representation does not provide sufficiently convenient access to neighborhoods. To determine which edges or faces are incident to a vertex, the corresponding lists must be searched.

1.2.1. Winged Edges. This representation uses lists (tables) of vertices, edges, and faces. The essential information about the mesh is stored in the edge list, while the other two structures are auxiliary. Each edge is assigned an orientation, and the following information is stored for it:

- the start vertex (a pointer to it or another identifier)
- the end vertex
- the left face
- the right face
- the left traversal:
 - the edge of the left face incident to the start vertex

- the edge of the left face incident to the end vertex
- the right traversal:
 - the edge of the right face incident to the start vertex
 - the edge of the right face incident to the end vertex

For a static structure, a unique edge identifier is also stored. Each edge is therefore stored in the table as an oriented line segment. This orientation is unrelated to the orientation of the mesh and serves only to order the data in the edge record.

For every vertex, the vertex list contains a pointer to a mesh edge incident to that vertex. If several such edges exist, the pointer may refer to any of them. Similarly, the face list stores, for each face, a pointer to one of its boundary edges.

1.2.2. DCEL (Doubly Connected Edge List). This structure can represent only orientable surfaces. We first orient the entire polygon mesh. Next, each interior edge is replaced by a pair of *half-edges*. These are two oppositely oriented line segments that would both coincide with the original edge if their orientations were ignored. Each half-edge then belongs to only one face, namely the face in which its orientation indicates the order of the vertices. Edges on the mesh boundary are replaced by half-edges with the appropriate orientation.

As in the previous case, a half-edge list contains all information about the mesh structure. Assume that the mesh faces have been oriented counterclockwise. The list then records the following information for each half-edge:

- a pointer to the start vertex
- a pointer to the face to the left of the half-edge (the face to which the half-edge belongs)
- a pointer to the opposite half-edge
- a pointer to the half-edge leaving the end vertex and continuing the boundary of the left face.

In addition to the half-edge list, there are two auxiliary structures: a vertex list and a face list. For each vertex, the vertex list contains a pointer to one of the half-edges leaving it. For each face, the face list contains one of the half-edges forming its boundary.

Both winged-edge and half-edge structures provide fast access to neighborhoods, which is a major advantage for meshes with very large numbers of vertices. They therefore support efficient representations of dynamic meshes in which not only vertex positions but also the mesh topology change over time. On the other hand, they are more complicated than the simple mesh representation introduced above. In some applications, the advantages of winged edges and half-edges are not used at all. One example is a mesh created only to visualize an object originally represented in another way. In such a case, a simpler representation is naturally preferable.

These structures also restrict the meshes that they can represent. In both cases, the structures are used only for orientable manifolds. However, they are only basic data structures and can be modified for a particular application to represent some nonmanifold surfaces as well.

1.3. Acquiring Data for a Polygon Mesh. Entering data for this representation directly is practical only for very simple objects. Meshes are clearly intended primarily as an internal system representation and are not suitable for communication with the user. Polygon meshes are often obtained by converting other types of representations, as we will see later. We now describe several special constructions.

1.3.1. Using a Parametrization. First, observe that a parametric surface can be approximated by a polygon mesh very easily. A *parametric surface* is a vector-valued function of two variables

$$f: D \rightarrow \mathbb{R}^3 \quad (D \subset \mathbb{R}^2), \quad (u, v) \mapsto (f_x(u, v), f_y(u, v), f_z(u, v)),$$

and suppose, in addition, that the function is injective, so that the resulting surface has no self-intersections. Divide the domain D into rectangles. For example, if D is the Cartesian product

of the intervals $\langle u_{min}, u_{max} \rangle \times \langle v_{min}, v_{max} \rangle$, divide each interval into several subintervals with endpoints

$$\begin{aligned} u_{min} &= u_0 < u_1 < \cdots < u_n = u_{max}, \\ v_{min} &= v_0 < v_1 < \cdots < v_m = v_{max}. \end{aligned}$$

This produces a grid of rectangles with vertices $(u_{i-1}, v_{j-1}), (u_i, v_{j-1}), (u_i, v_j), (u_{i-1}, v_j)$. The images of the rectangle vertices then form the vertices of the “quadrilaterals” in the mesh representation of the surface. It is often appropriate to split these “quadrilaterals” into two triangles because, in general, the images of the four vertices of a rectangle need not be coplanar.

Example 1.6. Represent the following parametric surface by a polygon mesh:

$$x = u, \quad y = v, \quad z = \frac{u+v}{2} - uv,$$

where $u \in \langle 0, 1 \rangle, v \in \langle 0, 1 \rangle$.

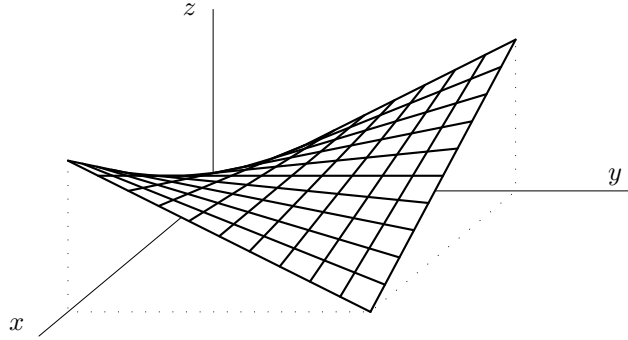


FIGURE 1. Parametric surface

Divide the intervals in both the u and v directions to obtain a grid of rectangles. Compute the coordinates of the images of all vertices, which gives the coordinates of the polygon-mesh vertices. It remains only to connect the vertices into polygons. In the illustrated example, the surface is rendered using “quadrilaterals,” but these shapes are not planar. Therefore, before applying more complex algorithms to the object, it is appropriate to split each “quadrilateral” into triangles.

There are many ways to obtain a parametric surface. We mention at least the simplest ones.

1.3.2. Translational Surfaces. A surface is obtained by translating a selected curve through space. The resulting surface is the union of the trajectories of all points of the curve. The curve may be translated along a straight line or, more generally, along another curve. Let the curve C , which is to be translated, be given parametrically by

$$(1) \quad x = c_x(u), \quad y = c_y(u), \quad z = c_z(u),$$

where u is a parameter ranging over the interval $\langle u_0, u_1 \rangle$. Let the translation trajectory be a curve T , also given parametrically:

$$x = t_x(v), \quad y = t_y(v), \quad z = t_z(v), \quad v \in \langle v_0, v_1 \rangle,$$

so the vector $(t_x(v) - t_x(v_0), t_y(v) - t_y(v_0), t_z(v) - t_z(v_0))$ is the translation vector at “time” v . The resulting surface is therefore parametrized by

$$\begin{aligned} x &= f_x(u, v) = c_x(u) + t_x(v) - t_x(v_0) \\ y &= f_y(u, v) = c_y(u) + t_y(v) - t_y(v_0) \\ z &= f_z(u, v) = c_z(u) + t_z(v) - t_z(v_0), \quad u \in \langle u_0, u_1 \rangle, v \in \langle v_0, v_1 \rangle. \end{aligned}$$

1.3.3. *Surfaces of Revolution.* When constructing a *surface of revolution* from the curve (1), we describe the surface generated by rotating the curve about an axis. Instead of giving general formulas, consider a specific example. We describe a torus as the surface generated by rotating a circle in the xz -plane about the z -axis, as shown in the figure. The circle in the xz -plane is

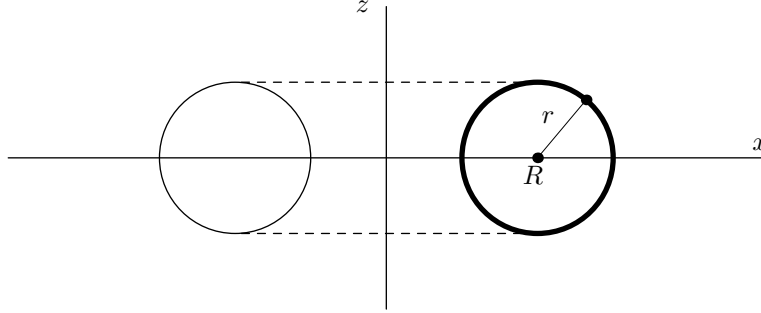


FIGURE 2. Construction of a torus

the parametric curve

$$x = R + r \cos u, \quad y = 0, \quad z = r \sin u,$$

where r is the radius of the circle, R is the distance from its center to the origin, and u is a parameter ranging over the interval $\langle 0, 2\pi \rangle$. We rotate every point of this curve about the z -axis. Substituting the curve parametrization into the rotation equations gives the following parametrization of the surface:

$$\begin{aligned} x &= (R + r \cos u) \cos v \\ y &= (R + r \cos u) \sin v \\ z &= r \sin u, \end{aligned} \quad \text{where } u, v \in \langle 0, 2\pi \rangle.$$

1.3.4. *Sweeping.* To describe a surface by sweeping, we start with a planar curve and move it through space along a prescribed trajectory. The plane containing the curve is usually kept perpendicular to the instantaneous direction of motion. The resulting surface is the union of the trajectories of all points of the curve. One degree of freedom remains during this motion: rotation within the plane of the curve about its intersection with the trajectory. The curve is also usually allowed to change its shape continuously during the motion.

Sweeping is therefore a fairly general way to describe a surface. It is sometimes regarded as a representation method in its own right. In other cases, sweeping serves as a user-interface representation that is subsequently converted into a polygon mesh inside the system.

1.3.5. *Procedural Modeling.* Among the many specialized methods for generating objects, consider the use of Brownian motion to generate terrain. The goal is to obtain a polygon mesh that represents rugged terrain as realistically as possible.

The starting point may be a flat horizontal triangular surface. In the first iteration of the algorithm, the coordinates of the triangle vertices are changed randomly. In the next step, a new vertex divides each triangle edge into two parts, thereby dividing the original triangle into four smaller triangles. The coordinates of the new vertices are then changed randomly. The process iteratively refines the mesh and randomly displaces the vertices. A rectangle may be used instead of a triangle and subdivided into four parts at each step.

The method used to displace vertices depends on the application. To obtain uniformly rugged terrain, the same random-number generator is used throughout. The displacement direction must also be chosen. The simplest approach changes only the height of a point and leaves the other two coordinates unchanged, as shown in the figure. However, the resulting terrain may be unsatisfactory. For example, this method can never model rock overhangs. To create such features, a new point on a triangle edge should be displaced in a direction perpendicular to the original surface.

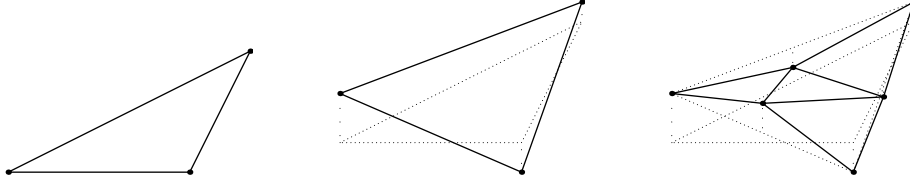


FIGURE 3. Terrain generation

This introduces another problem. If the point lies on an edge shared by two adjacent triangles, there are two different directions in which the point could be displaced. (For more illustrations, see, for example, <http://local.wasp.uwa.edu.au/~pbourke/fractals/planets/>).

1.3.6. Reconstruction of Real-World Data. Data for an object representation are often acquired by copying a real object rather than by modeling it artificially. Various 3D scanners can scan an object and output the coordinates of many points on its surface. The remaining task is to connect these points with edges to form a polygon mesh, using methods from discrete geometry. If some part of the object has been scanned poorly, the resulting holes may be filled with additional points using geometric-modeling methods.

Another approach reconstructs three-dimensional data from two-dimensional images, as in photogrammetry and computer vision. For example, suppose we have two projections, or two photographs taken from different sides, of the same object. The user identifies several key points in the images, such as some corners of a box, after which the system attempts to reconstruct the entire object automatically. If several projections of the object are available, a higher degree of automation is naturally possible. Conversely, we may want to reconstruct an object for which only one projection is available. In that case, additional information is required, such as knowledge about the object's symmetry or lighting.

1.4. Modifying Polygon Meshes. Modifying an object represented by a polygon mesh requires, in the better case, changing the coordinates of a very large number of vertices. In the worse case, the mesh topology, meaning the way vertices are connected by edges and faces, must also change. Such modifications require a degree of independent judgment and creativity in each particular case.

The simplest way to modify an object is to use affine transformations. An object can easily be translated, rotated, or scaled by applying the corresponding affine transformation to every vertex of the polygon mesh. This tool is often insufficient, however. To apply a different deformation to an object, suitable transformation equations must be designed.

Example 1.7. Suppose we have modeled a cuboid with, for example, a height of 5 cm, a width of 1 cm, and a thickness of 0.2 cm, with the vertical z -axis passing through its center. We want to deform it into a screw-like shape by rotating it about the z -axis. The higher a point of the cuboid lies, meaning the greater its z -coordinate, the larger the rotation angle. The twisting equations are

$$\begin{aligned}x' &= x \cos(\varphi(z)) - y \sin(\varphi(z)) \\y' &= x \sin(\varphi(z)) + y \cos(\varphi(z)) \\z' &= z,\end{aligned}$$

where $\varphi(z)$ is an increasing function, for example $\varphi(z) = z$ (the input is in centimeters and the output is in radians). It remains only to represent the cuboid by a sufficiently dense polygon mesh and apply the transformation equations to every vertex.

Scaling can be applied similarly when an object should be narrowed in the x and y directions, but only in a particular region. The standard scaling equations are used, except that the scaling

factor, like the angle in the previous example, is not constant and depends on the height of the point.

KAGDM FMFI UK BRATISLAVA

Email address: `jana.pilnikova@fmph.uniba.sk`